OCR Computer Science A Level

1.2.4 Types of Programming Language

Advanced Notes

Specification:

1.2.4 a)

- **Programming paradigms**
  - Need for these paradigms
  - Characteristics of these paradigms

1.2.4 b)

- **Procedural languages**

1.2.4 c)

- **Assembly language**
  - Following LMC programs
  - Writing LMC programs

1.2.4 d)

- **Modes of addressing memory**
  - Intermediate, Direct, Indirect, Indexed

1.2.4. e)

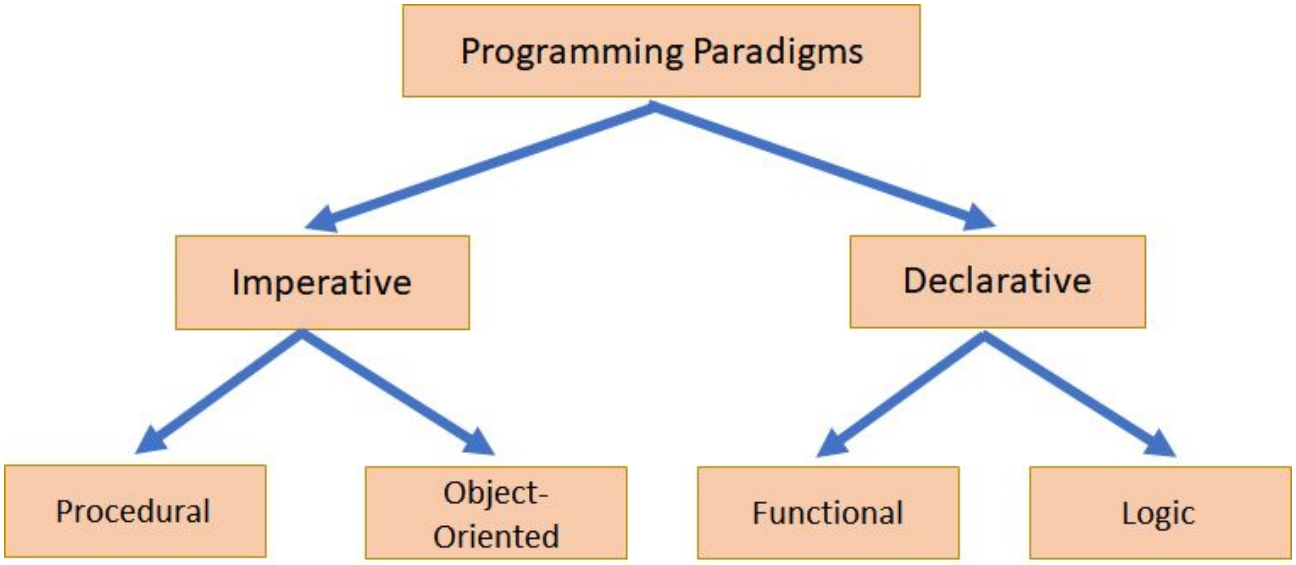
- **Object-oriented languages**
  - Classes
  - Objects
  - Methods
  - Attributes
  - Inheritance
  - Encapsulation
  - Polymorphism

Programming Paradigms

Programming paradigms are different approaches to using a programming language to solve a problem. They are split into two broad categories - imperative and declarative - which can be broken down further into more specific paradigms. The imperative programming paradigm includes the procedural and object-oriented paradigms while the declarative paradigm is split into logic and functional paradigms. The paradigm used depends on the type of problem that needs solving.

Note

Most programming languages such as Python allow for more than one paradigm.



Imperative

Imperative programming paradigms use code that clearly specifies the actions to be performed.

Procedural

Procedural programming is one of the most widely-used paradigms as it can be applied to a wide range of problems and is relatively easy to write and interpret. This is a type of imperative programming which uses a sequence of instructions which may be contained within procedures. These instructions are carried out in a step-by-step manner.

Synoptic Link

You will encounter the programming constructs discussed here again in 2.2.

Examples: *Pascal, Python, Logo*

Object-Oriented

Object-oriented programming (referred to as OOP) is another popular paradigm as it is applicable to certain types of problem with lots of reusable components which have similar characteristics. OOP is built on entities called objects formed from classes which have certain attributes and methods. OOP focuses on making programs that are reusable and easy to update and maintain.

Examples: *Python, Delphi, Java*

Declarative

Declarative programming focuses on stating the desired result rather than the exact series of instructions that need to be performed to get to the result. It is the role of the programming language to determine how best to obtain the result and the details about how it is obtained are abstracted from the user. This type of programming is common in expert systems and artificial intelligence.

Synoptic Link

Declarative languages and OOP are built around the principle of abstraction, which is explored in 2.1.

Function

A named block of code that performs a specific task and returns a value.

Functional

Functional programming uses the concept of reusing a set of functions, which form the core of the program. Programs are made up of lines of code consisting of function calls, often combined within each other. Functional programming is closely linked to mathematics.

Examples: *Haskell, C#, Java*

Logic

Logic languages are also part of the declarative programming paradigm and use code which defines a set of facts and rules based on the problem. Queries are used to find answers to problems.

Example: *Prolog*

Procedural Language

Procedural programming is used for a wide range of software development as it is very simple to implement. However, it is not possible to solve all kinds of problems with procedural languages or it may be inefficient to do so.

Procedural languages use traditional data types such as integers and strings which are built into the language and also provide data structures like dictionaries and arrays.

Structured programming is a popular subsection of procedural programming in which the control flow is given by four main programming structures:

- Sequence
  Code is executed line-by-line, from top to bottom.
- Selection
  A certain block of code is run if a specific condition is met, using IF statements.
- Iteration
  A block of code is executed a certain number of times or while a condition is met. Iteration uses FOR, WHILE or REPEAT UNTIL loops.
- Recursion
  Functions are expressed in terms of themselves. Functions are executed, calling themselves, until a certain condition known as a base case (which does not call the function) is met.

Therefore procedural programming is suited to problems that can easily be expressed as a series of instructions using the constructs described above.

Assembly Language

Assembly language is the next level up from machine code and is part of a family of low level languages. This is converted to machine code using an assembler when it is executed.

Mnemonic

An abbreviation for a machine code instruction in assembly code.

Assembly language uses mnemonics rather than binary, which makes it easier to use than direct machine code. Each mnemonic is represented by a numeric code. However, the commands that assembly language uses are processor-specific as it directly

interacts with the CPU's special purpose registers. This allows for direct interaction with hardware so is useful in embedded systems.
Typically, each instruction in assembly language is equivalent to almost one line of machine code.

Below is a list of the mnemonics you need to be aware of and be able to use:

Mnemonic	Instruction	Function
ADD	Add	Add the value at the given memory address to the value in the Accumulator
SUB	Subtract	Subtract the value at the given memory address from the value in the Accumulator
STA	Store	Store the value in the Accumulator at the given memory address
LDA	Load	Load the value at the given memory address into the Accumulator
INP	Input	Allows the user to input a value which will be held in the Accumulator
OUT	Output	Prints the value currently held in the Accumulator
HLT	Halt	Stops the program at that line, preventing the rest of the code from executing.
DAT	Data	Creates a flag with a label at which data is stored.
BRZ	Branch if zero	Branches to a given address if the value in the Accumulator is zero. This is a conditional branch.
BRP	Branch if positive	Branches to a given address if the value in the Accumulator is positive. This is a conditional branch.
BRA	Branch always	Branches to a given address no matter the value in the Accumulator. This is an unconditional branch.

Below is an example of an LMC program which returns the remainder, called the modulus, when num1 is divided by num2.

```
INP
STA num1
INP
```

Note

In a high-level language, this program would be represented as a single instruction - MOD.

```
          STA num2
          LDA num1
positive  STA num1          // branches to the 'positive' flag,
          SUB num2          subtracting num2 while the result
          BRP positive      of num1 minus num2 is positive
          LDA num1
          OUT
          HLT
    num1  DAT
    num2  DAT
```

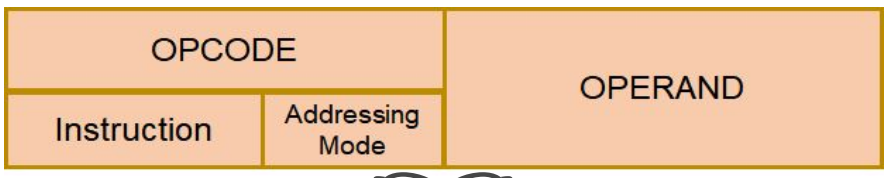
Modes of Addressing Memory

Machine code instructions are made up of two parts, the opcode and operand. The opcode specifies the instruction to be performed from the table above. The operand holds a value which is related to the data on which the instruction is to be performed.

In some cases, the operand may hold the actual value on which the instruction is to be executed but more often, it holds an address related to where this data is stored. Addressing modes allow for a much greater number of locations for data to be stored as the size of the operand would otherwise constrain the number of addresses that could be accessed.

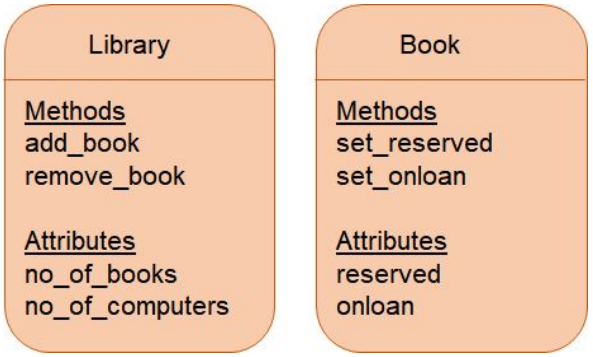
It is the addressing mode that specifies how the operand should be interpreted. The addressing mode is part of the opcode and there are four addressing modes you need to know:

- Immediate Addressing
  The operand is the actual value upon which the instruction is to be performed, represented in binary,
- Direct Addressing
  The operand gives the address which holds the value upon which the instruction is to be performed. Direct addressing is used in LMC.
- Indirect Addressing
  The operand gives the address of a register which holds another address, where the data is located.
- Indexed Addressing
  An index register is used, which stores a certain value. The address of the operand is determined by adding the operand to the index register. This is necessary to add an offset in order to access data stored contiguously in memory such as in arrays.



Object Oriented Language

Object-oriented languages are built around the idea of classes. A class is a template for an object and defines the state and behaviour of an object. State is given by attributes which give an object's properties. Behaviour is defined by the methods associated with a class, which describe the actions it can perform.
Classes can be used to create objects by a process called instantiation. An object is a particular instance of a class, and a class can be used to create multiple objects with the same set of attributes and methods.
A class is usually associated with an entity. For example, take a class called `'Library'`. It could have attributes '`number_of_books`', '`number_of_computers`' and methods '`add_book`' and '`remove_book`'. Similarly, 'Book' could also be a class.



`set_reserved` and `set_onloan` are a special type of method called Setters. A setter is a method that sets the value of a particular attribute. In this example, '`set_reserved`' would set the attribute '`Reserved`' to '`True`' if someone was to reserve that book. A getter is another special method used in OOP which retrieves the value of a given attribute.

The reason getters and setters are used is to make sure attributes cannot be directly accessed and edited by users. This property of object-oriented programming is called encapsulation. Attributes are declared as private so can only be altered by public methods. Every class must also have a constructor method, which is called '`new`'. A constructor allows a new object to be created.

Below is part of the pseudocode for the 'Book' class described above:

```
class Book:
     private reserved
     private onLoan
     private author
     private title
     public procedure new(title,author,reserved,onLoan)
          title = givenTitle
          author= givenAuthor
          reserved = givenReserved
          onLoan = givenOnLoan
     end procedure
     public function set_reserved()
```

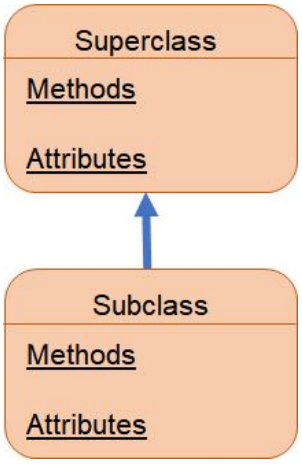
```
            reserved=True
        end function
end class
```

In order for a new object to be created, the constructor is used to define the characteristics of the object as shown below:

```
myBook = new Book('Great Expectations', 'Charles
Dickens','False',False')
```

When a book is reserved, the following code would be used to call the setter function on the myBook object.

```
myBook.set_reserved()
```



Another property of object-oriented programming is inheritance. A class can inherit from another class and this relationship between classes is shown using the diagram to the right. The subclass (or derived class) will possess all of the methods and attributes of the superclass (or parent class) and can have its own additional properties. Considering the previous example, the 'Biography' class could inherit from 'Book' and have its own attributes such as 'Subject' and a shorter loan duration, but would still have an author and title, for example. This allows programmers to effectively reuse certain components and properties while making some changes.

Inheritance would be expressed as:

```
    class Biography inherits Book
```

Polymorphism is a property of OOP that means objects can behave differently depending on their class. This can result in the same method producing different outputs depending on the object involved. There are two categories of polymorphism: overriding and overloading.

Overriding is redefining a method within a subclass and altering the code so that it functions differently and produces a different output.
Overloading is passing in different parameters into a method. Both of these forms of polymorphism would produce different results which would depend on the scenario.

Advantages

- OOP allows for a high level of reusability, which makes it useful for projects where there are multiple components with similar properties. The properties of inheritance and polymorphism within OOP allow for this.
- Classes can also be used across multiple projects.

- Encapsulation is a key reason for choosing OOP as it makes the code more reliable by protecting attributes from being directly accessed. Code for different classes can also be produced independently of others.
- OOP requires advance planning to determine how the problem will be broken down into classes and how these will link to each other. A thorough design can produce a higher-quality piece of software with fewer vulnerabilities.
- The modular structure used in OOP makes it easy to maintain and update.
- There is a high level of abstraction and it is not necessary for programmers to know details about how code is implemented. Once classes have been created and tested, they can be reused as a black box which saves time and effort.

Disadvantages

- This is a different style of programming and so requires an alternative style of thinking. This can be difficult for programmers accustomed to other paradigms to pick up.
- OOP is not suited to all types of problems. Where few components are reused, OOP may in fact result in a longer, more inefficient program.
- Generally unsuitable for smaller problems.